

대한민국 특허청  
KOREAN INTELLECTUAL  
PROPERTY OFFICE

별첨 사본은 아래 출원의 원본과 동일함을 증명함.

This is to certify that the following application annexed hereto  
is a true copy from the records of the Korean Intellectual  
Property Office.

출원번호 : 10-2002-0076041  
Application Number PATENT-2002-0076041

출원년월일 : 2002년 12월 02일  
Date of Application DEC 02, 2002

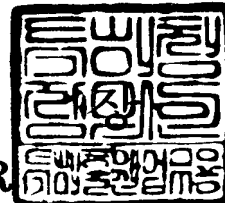
출원인 : 삼성전자 주식회사  
Applicant(s) SAMSUNG ELECTRONICS CO., LTD.



2002 년 12 월 21 일

특 허 청

COMMISSIONER



## 【서지사항】

【서류명】	특허출원서
【권리구분】	특허
【수신처】	특허청장
【참조번호】	0016
【제출일자】	2002.12.02
【국제특허분류】	G06F
【발명의 명칭】	자바 실행 장치 및 자바 실행 방법
【발명의 영문명칭】	A java execution device and a java execution method
【출원인】	
【명칭】	삼성전자 주식회사
【출원인코드】	1-1998-104271-3
【대리인】	
【성명】	이영필
【대리인코드】	9-1998-000334-6
【포괄위임등록번호】	1999-009556-9
【대리인】	
【성명】	이해영
【대리인코드】	9-1999-000227-4
【포괄위임등록번호】	2000-002816-9
【발명자】	
【성명의 국문표기】	정운교
【성명의 영문표기】	JUNG, Un Gyo
【주민등록번호】	720201-1454722
【우편번호】	442-711
【주소】	경기도 수원시 팔달구 매탄2동 원천성일아파트 202동 608호
【국적】	KR
【심사청구】	청구
【취지】	특허법 제42조의 규정에 의한 출원, 특허법 제60조의 규정에 의한 출원심사를 청구합니다. 대리인 이영필 (인) 대리인 이해영 (인)

**【수수료】**

【기본출원료】 20 면 29,000 원

【가산출원료】 18 면 18,000 원

【우선권주장료】 0 건 0 원

【심사청구료】 13 항 525,000 원

【합계】 572,000 원

【첨부서류】 1. 요약서·명세서(도면)\_1통

**【요약서】****【요약】**

본 발명에 따라 자바 실행 장치, 자바 클래스 파일 구조, 자바 실행 방법, 자바 파일을 사전 컴파일하는 방법, 자바 가상 머신에서 메소드를 실행하는 방법이 제공된다. 상기 본 발명에 따른 자바 실행 장치는 표준 클래스 라이브러리에 포함된 클래스 파일을 사전 컴파일하여 얻어진 머신 인스트럭션 클래스 파일을 포함하는 확장된 클래스 라이브러리와, 상기 확장된 클래스 라이브러리에 포함된 상기 머신 인스트럭션 클래스 파일 또는 어플리케이션 파일을 실행하는 자바가상머신을 포함한다. 이상과 같은 본 발명에 의하면 플랫폼 독립성과 동적 확장성을 보장하면서 성능향상을 가져올 수 있다.

**【대표도】**

도 7

**【명세서】****【발명의 명칭】**

자바 실행 장치 및 자바 실행 방법{A java execution device and a java execution method}

**【도면의 간단한 설명】**

도 1은 자바 프로그램이 실행되기 까지의 일반적인 과정을 설명하는 개념도,

도 2는 일반적인 자바의 계층 구조를 도시하는 개념도,

도 3은 종래기술에 따른 자바 플랫폼의 제1 예를 도시하는 도면,

도 4는 도 3에 도시된 자바 플랫폼의 제1 예에서의 동작과정을 나타내는 흐름도,

도 5는 종래기술에 따른 자바 플랫폼의 제2 예를 도시하는 도면,

도 6a는 본 발명에 따라 어플리케이션 소스 파일이 컴파일되는 과정을 나타내는 개념도,

도 6b는 본 발명에 따라 라이브러리 소스 파일이 컴파일되는 과정을 나타내는 개념도,

도 6c는 본 발명에 따른 m-클래스 파일의 머신 인스트럭션의 오퍼랜드에 들어있는 심볼 참조를 설명하는 도면,

도 7은 본 발명에 따른 자바 플랫폼의 구성의 일 예를 도시하는 도면,

도 8은 도 7에 도시된 자바 플랫폼에서의 동작 과정을 나타내는 흐름도,

도 9는 본 발명에 따라 클래스 파일을 입력받아 m-클래스 파일로 컴파일을 수행하는 과정을 나타내는 흐름도,

도 10은 본 발명에 따른 m-클래스 파일의 구조를 도시하는 도면,  
 도 11은 도 10에 도시된 mcode attribute 의 구조를 도시하는 도면,  
 도 12a는 본 발명에 따라 m-클래스 파일의 머신 인스트럭션 오퍼랜드에 들어있는  
 공통 심볼 참조 형식을 도시하는 도면,  
 도 12b는 도 12a에 도시된 심볼참조형식중 콘스탄트 풀 심볼 참조 형식을 도시하는  
 도면,  
 도 12c는 도 12a에 도시된 심볼참조형식중 JVM 내부 심볼 참조 형식을 도시하는 도  
 면,  
 도 12d는 도 12a에 도시된 심볼참조형식중 데이터 블록 위치 심볼 참조 형식을 도  
 시하는 도면,  
 도 13은 도 12b에 도시된 콘스탄트 풀 심볼 종류를 도시하는 도면,  
 도 14는 도 12c에 도시된 JVM 내부 심볼 지시자를 도시하는 도면,  
 도 15는 종래 orp 플랫폼과 본 발명에 따른 m-orp 플랫폼의 성능 결과를 비교하기  
 위한 실험 결과를 도시하는 도면,

#### 【발명의 상세한 설명】

#### 【발명의 목적】

#### 【발명이 속하는 기술분야 및 그 분야의 종래기술】

<21> 본 발명은 자바 플랫폼에 관한 것으로, 좀더 구체적으로는, 자바 실행 장치, 자바 클래스 파일 구조, 자바 실행 방법, 자바 파일을 사전 컴파일하는 방법, 자바 가상 머신에서 메소드를 실행하는 방법에 관한 것이다.

- <22> 자바 언어는 다양한 마이크로웨이브 오븐과 원격 제어기 같은 다양한 전자 장치 및 전자 제품에 삽입될 소프트웨어에 사용될 수 있는 플랫폼 독립적인 언어의 필요성에 의해 SUN 사에 의해 개발되었다.
- <23> 자바는 플랫폼 독립적인 실행 파일을 생성하기 위해 소스를 자바만의 독특한 바이트코드 형식으로 컴파일하고 자바 인터프리터를 이용하여 수행시킨다. 도 1에 도시된 바와 같이 .java 형식의 자바 프로그램(110)은 컴파일러(120)에 의해 컴파일되어 .class 의 자바 실행 파일로 변환되고, 이러한 클래스 파일은 인터프리터(130)에 의해 인터프리트되어 실제 실행된다. 인터프리터는 프로그램 수행에 필요한 모든 클래스들을 로드하는 클래스 로딩, 클래스 파일 포맷, 접근 권한, 데이터 형변환 등에 대한 검사를 수행하는 검증, 실제 프로그램 실행의 3단계로 진행된다.
- <24> 도 2에 자바의 계층 구조를 도시한다. 상기 계층 구조는 자바 언어로 작성된 자바 프로그램(240)과, 자바 가상 머신(220) 및 자바 API(230)를 포함하는 자바 플랫폼과, 하드웨어 의존적 플랫폼(210)을 포함한다. 이와 같은 구조 하에서 자바의 실행 파일은 특정 플랫폼과는 무관한 ByteCode 이므로 어느 시스템에서 개발했는지에 상관없이 자바 수행 환경(JRE)만 있으면 수행가능하다.
- <25> 자바 기술은, WORA(Write Once Run Anywhere)를 보장하는 플랫폼 중립성, 동적확장성 등 많은 장점으로 인해 여러 곳에서 사용되고 있다. 웹 서비스를 위한 서버 기술로 활발히 사용되어 대부분의 웹 어플리케이션 서버가 자바 기술을 기반으로 하고 있으며, 임베디드 기기에서는 사용자 서비스를 제공하거나 제어 어플리케이션을 수행하기 위한 환경으로서 자바 기술을 채용하는 예가 많아지고 있다. 특히, 이동전화를 위한 ME<sub>x</sub>E, 디지털 TV를 위한 MHP, DASE, OCAP 등은 임베디드 기기의 어플리케이션 환경을 자바 기

반 환경으로 정의한 표준 규격들로서 향후 임베디드 기기 시장에 대한 자바의 전망을 밝게 하는 근거라 할 수 있다.

<26> 그러나, 자바 기술의 쓰임새는 점점 더 넓어지고 있지만 자바 기술을 사용할 경우 자바 어플리케이션의 성능이 네이티브 어플리케이션과 비교할 때 만족할 만한 성능을 보이지 못하는 단점이 자바 기술 확산에 큰 걸림돌로 작용해왔다.

<27> 최근 성능 문제를 해결하기 위한 여러 가지 방법이 강구되어 왔고 그 결과 어느 정도의 성과를 거두어 왔다. 이 방법들은 주로 메소드의 바이트코드를 머신 코드로 컴파일함으로써 기존 자바 가상머신에서 사용하던 인터프리팅 방식의 비효율성을 개선하려는 것인데, 이를 다음과 같은 세가지 유형으로 분류할 수 있다.

<28> 첫 번째는 Just-In-Time(JIT) 컴파일 방식으로서, 자바 가상 머신이 자바 어플리케이션을 실행하다가 메소드 호출 시점에 호출되는 메소드를 컴파일을 하여 바이트코드 대신 컴파일된 머신 코드를 직접 실행시킨다.

<29> 이 방법은 인터프리팅 방식에 비해 상당한 속도 향상을 가져올 수 있지만 어플리케이션이 사용하는 메모리 이외에 JIT를 위해 수 MB 이상의 램이 추가적으로 요구된다. 이는 JIT 컴파일을 수행하기 위해서도 상당한 양의 메모리가 필요할 뿐만 아니라, 메소드를 컴파일하여 얻은 머신 코드가 다음 사용을 위해 메모리에 유지되어야 하기 때문이다. 또한, 이 방법은 어플리케이션 실행중에 호출되는 모든 메소드를 컴파일하므로 컴파일에 소요되는 실행 중 오버헤드가 크다. 이러한 유형에 해당하는 JVM의 예는 인텔에서 연구용으로 개발하여 공개한 ovp를 들 수 있다.

<30> 두 번째 유형은 동적 적응성(dynamic adaptive) 컴파일 방식이다. 이 방식은 JIT 컴파일과 인터프리터를 함께 사용하는 방식으로, 첫 번째 유형에서처럼 모든 메소드를 컴파일하는 것이 아니라, 성능에 많은 영향을 미치는 중요한 메소드(HOT 메소드)에 대해서만 컴파일하고 그렇지 않은 경우는 인터프리터를 통해 수행한다. 핫 메소드를 판단하기 위하여 어플리케이션을 실행하면서 프로파일링이 이루어지는데 핫 메소드의 판단 기준은 호출되는 횟수가 일정 기준을 넘으면 핫 메소드로 간주하는 단순한 방법을 비롯하여 여러 가지가 사용될 수 있다. 도 3은 이러한 유형의 방식을 사용하는 자바 플랫폼(300)의 일반적인 구조이다. 상기 자바 플랫폼(300)은 클래스 라이브러리(320)와, 자바 가상머신(330)을 포함하는데, 자바가상머신(330)은 JIT 컴파일러(340)와, JIT 컴파일을 하지 않는 메소드를 실행하는 인터프리터(350)와, 클래스 파일로부터 필요한 클래스를 로드하는 클래스로더(360)와, 메소드 영역, 자바 스택 등 실행 중 필요한 자료구조를 유지하고 전체 요소들을 결합 및 운영하는 런타임 시스템(370)을 포함한다.

<31> 도 4는 이러한 유형의 JVM에서 메소드 하나를 실행하기 위한 전형적인 실행 흐름을 보여준다. 메소드가 호출되면(S410) 먼저 그 메소드가 이전에 JIT 컴파일되어 머신 코드를 가지고 있는지 확인한다(S420). 만일 이미 컴파일된 메소드이면 바로 그 머신 코드를 실행한(S460) 후 리턴하지만 그렇지 않다면 호출된 메소드의 프로파일 정보를 가져와서 갱신한 뒤(S430), 호출된 메소드가 핫 메소드인지 그렇지 않은지를 프로파일 정보를 근거로 판단한다(S440). 핫 메소드이면 JIT 컴파일러에게 메소드의 정보를 넘겨서 메소드의 바이트 코드를 컴파일하고(S450) 그 결과로 얻어진 타겟 머신 코드를 실행한다. 만일 핫 메소드가 아니면 인터프리터에게 메소드의 정보를 넘겨서 수행한다(S460). 메소드의 수행이 끝나면 메소드가 호출되기 이전 지점으로 복귀한다. 메소드의 수행 도

중 다른 메소드를 호출하는 경우, 호출되는 메소드에 대해서도 이 흐름도가 마찬가지로 적용된다.

<32> 이 방법을 사용하는 JVM은 실행되는 모든 메소드들중에서 일부 메소드만 컴파일하므로 첫 번째 유형에 비해 컴파일로 인한 시간지연이 적으며 또한 유지해야할 머신 코드의 양이 적으므로 메모리 부담도 상대적으로 적다. 그러나, 핫 메소드가 아닌 메소드는 인터프리팅하여 실행하므로 JIT 컴파일 뿐만 아니라 인터프리터도 함께 가지고 있어야 하는 부담이 있고, 핫 메소드를 판단하기 위한 프로파일링이 실행 중 오버헤드로 작용한다. 상대적인 장단점은 있으나 메모리 크기가 제한적인 임베디드 기기에서는 일반적으로 첫 번째 유형보다 두 번째 유형이 사용된다. 이 두 번째 유형에 해당하는 예는 SUN의 CVM과 Insignia의 Jeode 등이 있다.

<33> 세 번째 유형은 Ahead-Of-Time(AOT) 컴파일 방식이다. JIT 컴파일러는 JVM의 내부에 포함되어 어플리케이션의 실행중에 동작하지만, AOT 컴파일러는 JVM과 별도로 존재하고 사용된다. JIT 컴파일러는 JVM 내부에 포함되어 어플리케이션의 실행중에 동작하지만, AOT 컴파일러는 JVM과 별도로 존재하고 사용된다. AOT 컴파일러는 어플리케이션 개발 환경에서 사용되며, 일반적으로 자바 클래스 파일을 컴파일하여 타겟 기기에서 수행될 수 있는 실행 파일을 생성한다. 도 5는 AOT 컴파일러를 사용하는 일반적인 과정을 나타낸다.

<34> 자바 소스 파일이나 클래스 파일 형태의 어플리케이션 파일(510)은 AOT 컴파일러(520)를 통해 컴파일되어 어플리케이션이 실행될 타겟 기기에 맞는 오브젝트 파일(540)로 변환된다. 이 때 어플리케이션 실행을 위해 필요한 라이브러리 클래스들(530)도 함께 컴파일된다. 오브젝트 파일은 링커(550)에 의해 런타임 시스템 모듈(560)과 링크됨

으로써 타겟 기기에서 독립적으로 실행될 수 있는 실행파일(570)이 만들어진다. 런타임 시스템 모듈은 JVM의 여러 기능중 바이트코드 실행 엔진을 제외한 나머지 기능을 제공하기 위한 모듈로서 쓰레기 수집, 타입 리플렉션 등의 기능을 제공한다.

<35> 이러한 세 번째 유형은 앞서 설명한 유형들과 접근법에서 뚜렷한 차이가 있다.

AOT 컴파일 방식은 자바 언어로 작성된 프로그램을 마치 C/C++ 로 작성된 경우와 유사하게 처리하여 타겟 환경에 종속적인 실행 파일을 만들어낸다. 앞의 두 유형은 어플리케이션을 배포할 때 자바의 표준 실행 파일 형식인 클래스 파일 형태로 배포하고, 타겟 기기에서 어플리케이션을 실행할 때 JVM 내에서 컴파일을 수행하지만 AOT 컴파일 방식은 개발 플랫폼에서 컴파일이 이루어지고, 컴파일을 통해 타겟 환경의 실행파일 형식으로 변환한 후 배포한다.

<36> 이와 같은 차이점에 기인하여, AOT 컴파일 방식을 사용하면 자바의 중요한 두가지 장점을 상실하게 된다.

<37> AOT 컴파일 방식을 사용할 때 잃게 되는 가장 중요한 특징은 플랫폼 독립성이다. 자바는 자바 가상 머신을 위한 실행코드, 즉 바이트코드를 지닌 클래스파일로 배포되므로 어떤 타겟 하드웨어 플랫폼에서도 자바 가상 머신에 의하여 실행될 수 있는 것인데, AOT 컴파일에 의해 특정 하드웨어에서만 실행 가능한 머신코드로 바뀌어지면 다른 기기에서는 실행될 수 없게 된다.

<38> 또 한가지 단점은 동적 확장 능력을 상실하게 된다는 것이다. 동적 확장은 실행중 새로운 타입을 인식하고 사용할 수 있는 기능으로서 C/C++과 비교하여 자바만이 가진 특별한 기능이다. 일반적인 AOT 컴파일 방식은 컴파일 당시에 어플리케이션 클래스와 어플리케이션이 사용하는 라이브러리 클래스들을 한꺼번에 컴파일하여 목적 파일을 생성

하는데; 어플리케이션 실행 도중에는 목적 파일에 포함된 클래스들 이외에 외부에서 새로운 클래스를 로드하여 실행할 수 없다.

<39> AOT 컴파일 방식은 앞서 설명한 바와 같이 자바의 중요한 특성을 상실한다는 단점이 있지만, 개발 환경에서 어플리케이션 배포 이전에 컴파일하므로 충분한 최적화 기법을 사용하여 빠른 성능을 가진 실행 파일을 생성할 수 있다. 이러한 이유로 타겟 실행 환경이 미리 정해지고, 실행 속도가 아주 문제가 되는 경우에 사용된다. GNU의 gcj는 이러한 AOT 컴파일러에 해당한다.

<40> 이와 같이 각 유형들은 각각의 장단점 및 성격이 다르므로 어떤 기기에 자바 플랫폼이 탑재될 때는 그 기기의 상황 및 용도에 맞추어 적당한 방법이 선택되는데, 성능 향상을 꾀하면서도 플랫폼 독립성과 동적 확장성을 잃지 않는 방법은 없다.

#### 【발명이 이루고자 하는 기술적 과제】

<41> 본 발명은 상기와 같은 과제를 해결하여 플랫폼 독립성과 동적 확장성을 보장하면서 성능향상을 가져올 수 있는 자바 실행 장치, 자바 클래스 파일 구조, 자바 실행 방법, 자바 파일을 사전 컴파일하는 방법, 자바 가상 머신에서 메소드를 실행하는 방법을 제공하는 것을 목적으로 한다.

#### 【발명의 구성 및 작용】

<42> 일반적으로, 자바 어플리케이션이 동작하기 위해 수행되는 코드는 어플리케이션 개발자가 작성한 어플리케이션 코드보다 자바 플랫폼에 포함된 클래스 라이브러리 코드가 훨씬 많으며 또한 훨씬 더 많은 수행 시간이 소요된다. 또한, 자바 어플리케이션 클래스는 어떤 기기에서도 수행될 수 있도록 자바 클래스 파일 형태로 배포되어야 하지만,

클래스 라이브러리는 일반적으로 JVM과 함께 특정 기기에 미리 설치되는 것이므로 하드웨어 종속적이라 하더라도 무방하다.

<43> 그러므로, 클래스 라이브러리에 대해서만 사전에 AOT 컴파일을 해두었다가 JVM에서 어플리케이션 수행시 컴파일된 라이브러리를 이용할 수 있다면 상당한 성능 향상을 기대할 수 있을 것이다. 또한, 자바 클래스 파일로 배포되는 어플리케이션은 JVM에서 인터프리팅 등의 방법으로 실행할 수 있으므로 어플리케이션의 플랫폼 독립성도 보장할 수 있다.

<44> 이와 같은 구현을 위해 본 발명은 자바 클래스 파일과 유사한 특성과 내용을 가진 m-클래스(modified class) 파일을 제안한다. 이러한 m-클래스 파일이 종래의 클래스 파일과 다른 점은 하드웨어 종속적인 바이트코드 대신 특정 타겟 기기의 머신코드를 포함한다는 것이다. 또한, 어플리케이션과 어플리케이션이 사용하는 전체 클래스들을 컴파일하던 기존 AOT 컴파일러와 달리 입력으로 주어진 클래스를 어플리케이션 클래스와 상관없이 컴파일을 하여 m-클래스 파일을 생성하는 AOT 컴파일러를 제안한다. 이렇게 함으로써, AOT 컴파일러를 이용하여 타겟 기기의 머신 코드로 변환된 클래스 라이브러리를 얻을 수 있다.

<45> 본 발명의 하나의 특징은, 자바 실행 장치에 있어서, 표준 클래스 라이브러리에 포함된 클래스 파일을 사전 컴파일하여 얻어진 머신 인스트럭션 클래스 파일을 포함하는 확장된 클래스 라이브러리와, 상기 확장된 클래스 라이브러리에 포함된 상기 머신 인스트럭션 클래스 파일 또는 어플리케이션 파일을 실행하는 자바가상머신을 포함하는 것이다.

- <46> 본 발명의 다른 특징은, 자바 클래스 파일 구조에 있어서, констан트와 필드와 메소드를 포함하며, 상기 메소드의 메소드 정보는 심볼 참조 정보가 삽입된 오퍼랜드를 포함하는 머신 인스트럭션으로 구성된 코드 속성을 포함하는 것이다.
- <47> 본 발명의 또다른 특징은, 자바 실행 방법에 있어서, a) 표준 클래스 라이브러리에 포함된 클래스 파일을 머신 인스트럭션을 포함하는 확장된 클래스 파일로 미리 사전컴파일하는 단계와, b) 상기 확장된 클래스 파일은 머신 인스트럭션을 실행하는 단계와, c) 자바 어플리케이션 파일은 JIT 컴파일 또는 인터프리트에 의해 실행하는 단계를 포함하는 것이다.
- <48> 본 발명의 또다른 특징은, 자바 파일을 사전 컴파일하는 방법에 있어서, 자바 클래스 파일 또는 자바 소스 파일을 심볼 참조 정보가 삽입된 오퍼랜드를 포함하는 머신 인스트럭션으로 변환하는 단계를 포함하는 것이다.
- <49> 본 발명의 또다른 특징은, 자바 가상 머신에서 메소드를 실행하는 방법에 있어서, 실행할 메소드의 메소드 정보는 심볼 참조 정보가 삽입된 오퍼랜드를 포함하는 머신 인스트럭션으로 구성된 코드 속성을 포함하는지를 판단하는 단계와, 상기 머신 인스트럭션으로 구성된 코드 속성을 포함하는 경우에, 상기 심볼 참조 정보를 직접 주소로 링크하여 상기 머신 인스트럭션을 실행하는 단계를 포함하는 것이다.
- <50> 이제, 첨부된 도면을 참조하여 본 발명을 상세히 설명한다.
- <51> 도 6a는 본 발명에 따른 어플리케이션 컴파일 과정을 도시하고, 도 6b는 본 발명에 따른 라이브러리 컴파일 과정을 도시한다.

- <52> 도 6a에 도시된 바와 같이, 자바 어플리케이션 소스 파일(.java)(610)은 자바 컴파일러(620)를 통해 컴파일되어 어플리케이션 클래스 파일(.class)(630)로 변환된다.
- <53> 그리고, 도 6b에 도시된 바와 같이, 라이브러리 소스(.java) 또는 클래스 파일(.class)(640)들은 사전 컴파일러(650)를 통해서 m-클래스 파일(660)로 변환된다.
- <54> 이와 같이 변환된 m-클래스 파일(660)은 머신 인스트럭션을 포함한다.
- <55> 도 6c에 도시된 바와 같이, m-클래스 파일(660)은 OP 코드(671)와 오퍼랜드(672)로 이루어진 머신 인스트럭션(670)으로 이루어진다. 그리고 머신 인스트럭션의 오퍼랜드(672)는 주소값을 가지는 것이 아니라 심볼 테이블(680)의 인덱스를 나타내는 심볼 참조값을 가진다.
- <56> 도 7은 본 발명에 따른 자바 플랫폼의 일 예를 도시한다.
- <57> 상기 자바 플랫폼(700)은 m-클래스 라이브러리(720)와 자바가상머신(730)을 포함한다. 그리고, 자바가상머신(730)은 m-클래스 링커(740)와, 인터프리터(750)와, 확장된 클래스 로더(760)와, 런타임 시스템(770)을 포함한다.
- <58> m-클래스 라이브러리(720)는 AOT 컴파일러로 변환된 m-클래스 파일들로만 이루어진 라이브러리이며, 이 라이브러리는 m-클래스 파일들과 표준 클래스 파일들이 섞여 있는 형태도 가능하다.
- <59> m-클래스 링커(740)는 m-클래스 라이브러리(720)에 들어있는 m-클래스 파일의 메소드의 mcode 속성 정보를 해석하여 실행가능한 완전한 머신코드로 변환하는 작업을 수행한다. 이 작업은 주로 AOT 컴파일러가 머신 코드에 폴딩한 각종 심볼참조들을 실제 주소값으로 바꾸는 작업이다. 또한, m-클래스 링커는 예외를 처리하거나 가비지 콜렉션을

수행해야 할 때 이에 필요한 정보를 mcode 정보에서 디코딩하여 JVM이 사용하는 자료구조로 바꾸어주는 일도 수행한다. 확장된 클래스로더(760)는 표준 자바 클래스 파일도 처리할 수 있을 뿐만 아니라 m-클래스 파일도 처리할 수 있도록 확장된 것이고, 인터프리터(750)는 종래 JVM의 인터프리터와 같은 것으로 m-클래스 파일로 미리 변환되지 않은 클래스 파일을 처리한 것이고 인터프리터 대신 JIT 컴파일러를 사용할 수도 있다. 런타임 시스템(770)은 종래의 JVM과 유사하나 m-클래스 파일을 처리할 수 있다.

<60> 어플리케이션 클래스(710)는 자바 가상 머신의 클래스 로더(760)에 의해 로드되어 런타임 시스템(770)내의 자료구조에 그 정보가 저장되고 어플리케이션의 첫 메소드인 메인 메소드가 m 클래스 링커(740) 또는 인터프리터(750)에게 넘겨져서 수행됨으로써 어플리케이션이 실행된다.

<61> 도 7에서 타겟 실행 기기에 미리 설치된 라이브러리가 m-클래스 파일들로 구성되어 있고 어플리케이션은 표준 자바 클래스 파일인 경우를 나타내었지만, 이러한 경우 외에도 표준 클래스 파일과 m-클래스 파일이 섞여 있는 라이브러리를 사용할 수도 있다. 또한 실행될 프로세서를 미리 알고 있는 경우에는 어플리케이션도 m-클래스 파일로 변환하여 배포할 수 있다. 도 7에서는 어플리케이션이 클래스 파일로 배포되는 경우를 나타내고 있으므로 메인 메소드가 인터프리터(750)에 넘겨져서 수행되지만, 어플리케이션이 m-클래스 파일로 배포되는 경우에는 메인 메소드가 m-클래스 링커에게 넘겨져 수행된다.

<62> 이제, 상기 m-클래스 라이브러리(720)에 포함된 m-클래스 파일의 생성 과정 및 구조를 설명한다.

<63> 도 9는 클래스 파일을 입력받아 m-클래스 파일로 컴파일을 수행하는 과정을 나타내는 흐름도(900)이다.

- <64> 본 발명에 따른 AOT 컴파일러는 클래스 파일을 입력으로 받아서 클래스의 각 메소드에 대해 독립적으로 컴파일을 수행한다.
- <65> 먼저, 메소드의 바이트코드 전체를 스캔하면서 지역 변수에 레지스터를 할당하기 위한 정보와 가비지 콜렉션을 위한 정보등을 수집하여 전처리한다(S910).
- <66> 다음, 전처리단계에서 얻은 정보를 바탕으로 지역변수에 레지스터를 할당한다(S920).
- <67> 그리고나서, 각 바이트 코드에 대응하는 타겟 머신의 인스트럭션 시퀀스를 생성한다(S930). 이때 인스트럭션의 오퍼랜드에는 주소값 대신 심볼 삽입 정보를 삽입한다.
- <68> 그리고, 코드 이미션 단계에서는 이전 단계에서 생성된 전체 인스트럭션들을 연속된 하나의 메모리 공간에 옮겨 담는다(S940).
- <69> 코드 생성단계에서는 전방 참조의 경우와 같이 패치가 필요한 인스트럭션들이 생성될 수 있는데 코드 및 데이터 패치 단계에서(S950)는 이러한 인스트럭션들의 패치 및 데이터 블록 내용의 패치가 이루어진다. 패치 단계에서도 인스트럭션내에 주소값 대신 심볼 참조 정보가 삽입되도록 한다. 이와 같은 AOT 컴파일 과정이 완료되면 머신 인스트럭션으로 이루어진 m-클래스 파일을 얻을 수 있다.
- <70> 이제, 도 10을 참조하여 m-클래스 파일의 구조를 설명한다.
- <71> m-클래스 파일은 AOT 컴파일된 결과를 저장하기 위한 본 발명에 따른 파일이다.
- <72> m-클래스 파일(1000) 형식은 JVM 규격에 정의된 표준 자바 클래스 파일 형식을 확장한 것이다. m-클래스 파일(1000)은 콘스탄트(1010)와, 필드(1020)와, 메소드(1030)를 포함하며, 메소드의 메소드 정보(1040)에는 mcode\_attribute(1050)를 포함한다. 이와

같이 메소드 정보 내부에 `code_attribute` 대신 `mcode_attribute(1050)`를 가지는 것이 표준 클래스 파일과 다른 점이다. 예를 들어, 각 메소드의 바이트코드를 저장하는 "code" 속성 대신에 AOT 컴파일 결과를 담기위한 "com.samsung.mcode" 속성을 포함한다. "com.samsung.mcode"라는 URL 형식의 이름은 클래스파일 규격에 명시한 사용자 정의 속성의 이름을 정하는 규칙을 따른 것이다. 본 발명에서는 m-클래스 파일의 크기를 줄이기 위하여 "code" 속성을 제거하였지만 만일 이를 제거하지 않는다면 m-클래스 파일은 표준 클래스 파일의 규격을 완전히 만족하게 된다. 즉, m-클래스의 특별한 정보를 인식하고 이용할 수 없는 JVM 이더라도 m-클래스 파일을 표준 클래스 파일로서 로드하여 실행할 수 있는 것이다.

<73> `mcode_attribute`는 표준 클래스 파일의 `code attribute`에 대응하는 것으로서 바이트 코드 대신 타겟 프로세서의 인스트럭션을 포함하고 있으며 또한 실행을 위한 기타 정보를 포함한다.

<74> 도 11은 `mcode_attribute`의 형식의 일 예를 도시하는데, 이러한 구조는 클래스 파일 규격의 속성 정보 형식을 따른다.

<75> 클래스 파일 규격의 `attribute` 형식에 따라 실행코드에 해당하는 부분과 데이터 정보를 저장하는 부분을 포함한다.

<76> `attribute name index(1051)`는 속성의 이름을 나타내고, `attribute length(1052)`는 속성 전체의 길이를 나타내며, `mcode length(1053)`는 뒤에 오는 `mcode`의 길이를 나타낸다.

- <77> mcode(1054)는 바이트코드가 AOT 컴파일러를 통해 변환된 타겟 프로세서의 인스트럭션이 저장되는 곳으로, 이러한 인스트럭션들은 오퍼랜드들에 대한 직접 주소 대신 심볼 참조를 사용하는 등 인스트럭션의 형식이 일부 수정된 것으로 실제 타겟 프로세서에서 실행되기 전에 심볼 결정 등의 처리가 필요한 코드이다.
- <78> data block length(1055)는 데이터 블록의 길이를 나타내고, data block(1056)는 부동형 소수값이나 분기 테이블 등이 저장된다.
- <79> symbolic reference list(1057)는 코드와 데이터에 포함된 심볼들의 위치 정보를 가진다.
- <80> exception handling info(1058)는 예외 처리를 위한 정보를 가지며, GC info(1059)는 가비지 콜렉션에 관한 정보를 가진다.
- <81> 본 발명에 따른 AOT 컴파일러는 메소드를 실행하기 위해서 필요한 머신 코드를 생성하고 또한 이 머신 코드가 실행중 참조하는 데이터 블록을 생성하는 것이 중요한 역할이다. 하지만 이것만으로는 바이트 코드의 도움 없이 완전한 기능을 수행할 수가 없고 추가적인 정보가 수집되어 m-클래스 파일에 저장되어야 한다.
- <82> 우선 예외처리를 위한 정보가 저장되어야 한다. 이것은 예외가 발생했을 때 JVM이 스택언와인딩을 하면서 이 예외를 처리하는 핸들러의 정확한 위치를 찾아 실행흐름을 이 동시켜주기 위해 필요하다.
- <83> 또한 가비지 콜렉션을 처리하기 위해 스택등에서의 타입 정보가 필요하다. 자바 바이트 코드는 오퍼랜드의 타입정보를 포함한 코드이지만 이것을 머신 코드로 바꾸고 나

면 타입 정보를 잃어버리는데 이 타입 정보는 GC가 발생하는 경우 알아야 하는 정보이기 때문이다. 이러한 타입 정보 등이 GC info(1059)에 저장된다.

<84>      이상 도 10과 도 11에 도시되어 있는 m-클래스 파일 구조와 mcode attribute 구조는 본 발명에 따른 AOT 컴파일 결과를 저장하기 위한 형식의 일 예이며, 본 발명이 이러한 형식에 제한되는 것은 아니라는 것을 당업자라면 충분히 이해할 것이다.

<85>      한편, m-클래스 파일이 자바 클래스 파일과 동일한 특성을 갖도록 하기 위해서는 모든 심볼들에 대해 동적 링크가 가능하도록 해야 한다. 자바의 이러한 특징은 바이트 코드에서의 객체의 필드 및 메소드를 가리키기 위하여 심볼 참조를 사용하기 때문에 가능한 특징이다. 그러므로 자바 클래스 파일을 AOT 컴파일하여 타겟 머신 코드로 변환하더라도 머신코드 인스트럭션의 오퍼랜드에 객체의 필드나 메소드를 가리키기 위하여 심볼 참조 방식을 사용한다면 자바의 동적 로드/링크 특징은 변함없이 구현가능하다.

<86>      m-클래스 파일에 포함된 타겟 머신 코드는 특정 필드나 메소드를 가리키기 위해 주소 대신 심볼 참조(Symbolic Reference)를 사용하는데, 심볼 참조 정보를 인스트럭션내에 삽입하기 위해 인스트럭션의 형식을 바꿀 수도 있다. 심볼 참조 정보가 포함된 인스트럭션들은 JVM에서 각 심볼에 해당하는 실제 주소값을 이용하여 패치되어 온전한 실행 코드로 바뀌어진다. JVM에서 m-클래스 파일을 인식하여 사용하기 위해서는 각각의 심볼 참조들을 실제 주소로 바꾸어주는 작업과 기타 몇가지 작업을 어플리케이션 수행 도중에 수행해야 한다. 하지만 이러한 작업은 JIT 컴파일에 비해 훨씬 간단한 작업이므로 이러한 기법을 이용한다면 JIT 컴파일하지 않고도 JIT 컴파일로 얻을 수 있는 머신코드보다 더 최적화된 머신 코드를 얻을 수 있고 결과적으로 더 우수한 성능을 낼 수 있다.

- <87> 본 발명에 따른 AOT 컴파일러는 모든 심볼 참조(1210)를 도 12a에 도시된 바와 같은 32비트 형식으로 인코딩하여 인스트럭션(1200)의 오퍼랜드 부분에 폴딩한다.
- <88> 32비트의 공통된 크기의 폴딩이 가능한 것은 x86 의 인스트럭션 형식에서 오퍼랜드가 주소값일 경우에는 32비트를 차지하기 때문인데 만일 다른 프로세서용 컴파일러를 구현할 경우에는 그 프로세서의 인스트럭션 규격에 맞추어 심볼 참조 형식을 고안해야 할 것이다.
- <89> 모든 심볼 참조의 공통 형식을 보면, 첫 두비트(1211)는 심볼 참조의 종류를 구분하는 플래그로 사용된다. 다음 14비트(1212)는 모든 심볼 참조들을 연결시키기 위한 링크로서 사용되는데 현재 심볼 참조 다음 바이트에서 다음 심볼 참조의 첫 바이트까지의 거리값을 가진다. 나머지 16비트(1213)는 심볼 참조가 가리키는 심볼이 무엇인지를 알아내기 위한 값이 저장된다.
- <90> 코드에서 사용되는 심볼들은 도 12b에 도시된 констан트 풀에 있는 심볼과, 도 12c에 도시된 JVM 내부의 심볼, 그리고 도 12d에 도시된 데이터 블록에서의 특정 위치 정보, 이 세가지로 구분된다.
- <91> констан트 풀은 클래스 파일에 포함된 일종의 심볼 테이블로서 자바 바이트코드에서 사용되는 모든 심볼들에 대한 정보를 가지고 있다. 자바 바이트코드는 객체의 필드나 메소드를 가리키기 위하여 오퍼랜드로서 констан트 풀 엔트리의 인덱스를 사용한다. 이와 마찬가지로 AOT 컴파일러에서도 오퍼랜드가 констан트 풀에 있는 클래스, 필드, 혹은 메소드 등일 경우에 이것들의 실제 주소값 대신 констан트 풀 인덱스(1223)를 가진 심볼 참조를 사용한다.

- <92>      콘스탄트 풀 엔트리를 보면 심볼의 종류를 알 수 있는데 AOT 컴파일러에서 사용되는 심볼의 종류는 클래스, 필드, 그리고 메소드의 세가지이며, 또한 각 종류마다 두가지 쓰임을 가지므로 도 13에 도시된 바와 같은 6가지의 심볼 참조가 생긴다.
- <93>      모든 JVM 내부 심볼 참조는 도 12c에 도시한 바와 같으며 내부 심볼 종류에 따라 16 비트 인디케이터(1223)가 인코딩된다. 도 14는 JVM 내부 심볼 지시자 구조를 도시한다. 첫 두 비트(1410)는 내부 심볼들의 종류를 구분하기 위한 플래그이며, 마지막 8비트(1420)는 내부 심볼들의 인덱스 값을 가진다.
- <94>      AOT 컴파일러가 생성하는 코드에서 사용되는 JVM 내부 심볼들은 여러 가지가 있으나 크게 다음 4가지로 구분할 수 있다. 첫째는 실행중 지원함수 인데, 이 함수들을 테이블화하여 얻을 수 있는 각 함수의 인덱스를 도 14의 심볼 인덱스 자리에 저장한다. 둘째는 사전 로드된 클래스로서 컴파일중인 클래스파일의 콘스탄트 풀에는 없지만, "java.lang.class" 클래스나 원시 타입 클래스 등과 같이 JVM 내에서 미리 로드되어 사용되는 클래스들을 말한다. 이 클래스들을 가리키기 위해서도 테이블화하여 얻은 인덱스를 사용한다. 셋째는 JVM이 가지고 있는 전역변수들이다. 넷째는 기타 심볼들이나 m-링커에게 알려줄 힌트를 저장하기 위해 사용된다.
- <95>      인스트럭션에서 데이터 블록의 특정 위치를 지정하는 경우에 일반적인 인스트럭션은 이 위치의 주소값을 오퍼랜드로 가지지만 AOT 컴파일러는 이 주소값을 심볼 참조로 대체한다. 이 심볼 참조의 형식은 도 12d에 나타나 있는데 도 12d에서 볼 수 있듯이 마지막 16비트(1243)가 데이터 블록의 시작 위치에서 해당 위치까지의 거리값을 저장하는데 사용된다.

- <96> 도 8은 본 발명에 따라 자바 플랫폼에서 메소드를 실행하는 과정(800)을 나타내는 흐름도이다.
- <97> 메소드가 호출되면(S810) 우선 이미 m-클래스 링커에 의해 링크되거나 링크되어 최초로 실행되는 것인지를 판단한다(S820).
- <98> 최초로 실행되는 것이 아니라면 이미 이전에 생성된 머신 코드가 있으므로 바로 이 머신 코드를 실행하면 된다(S860).
- <99> 최초로 실행되는 경우라면 이러한 메소드의 정보를 추출하여(S830) mcode 속성을 가진 메소드인지를 판단한다(S840). mcode 속성이 있다면 m-클래스 링커는 심볼 레졸루션 등의 작업을 처리하여 mcode를 링크하고(S850), 머신 코드를 실행한다(S860). 그리고, mcode 속성이 없다면 인터프리터에 의해 인터프리트된다(S870).
- <100> 이제, 도 15 및 도 16을 참조하여 본 발명에 따른 시뮬레이션 결과를 설명한다.
- <101> 도 15는 윈도우 XP 프로페셔널 운영체제가 설치된 펜티엄4 머신에서 실험한 orp와 본 발명에 따른 m-orp의 실행 속도 비교 결과를 도시한다. 이것은 헬로월드 어플리케이션을 각각 10회씩 반복 실행하여 평균 수치를 구한 것이다. FLT는 파일 로딩 타임을 나타내고, TT는 토탈 타임을 나타내고, JT는 JIT 컴파일 타임을 나타내고, MLT는 m-링킹 타임을 나타낸다.
- <102> 이 실험 결과에서 눈에 띄는 점은 파일 로드 시간이 전체 실행시간의 60% 이상을 차지한다는 점이다. 이 것은 실험환경인 PC의 프로세싱 속도가 매우 빨라 JIT 컴파일들의 처리작업에 소요되는 시간이 매우 적은 반면, 파일 로드를 위해 디스크 I/O를 하는 데에 상대적으로 많은 시간이 소요되기 때문이다. 그런데, 일반적으로 임베디드 기기는

디스크를 사용하지 않으므로 파일 I/O에 드는 시간의 비중이 도 15의 결과에서보다 훨씬 적다. 바꿔말하면 이 실험의 결과에서는 파일 I/O의 비중이 너무 커서 JIT 컴파일에 드는 시간과 m-링킹에 드는 시간의 차이가 전체 성능에 큰 영향을 주지 못하지만 임베디드 기기에서는 그 차이로 인해 전체 성능이 상당히 달라질 수 있다.

<103> 이 실험에서 JIT 컴파일은 m-링킹에 비해 4배 정도의 시간이 소요되었는데 이 차이가 자바 플랫폼의 전체 성능에 20%의 향상을 가져왔다. PC에서의 20% 성능향상도 의미있는 것이지만 임베디드 기기에 적용된 경우를 가정하여 파일 로드에서 소요된 시간을 제외하고 계산하면 80% 이상의 성능향상에 해당하는 것이다.

#### 【발명의 효과】

<104> 이상과 같은 본 발명에 의하면, JVM에서 메소드를 실행하기 위해 JIT 컴파일을 사용하던 기존의 방식을 사용할 때 보다 더 빠른 속도의 자바 프로그램을 실행할 수 있다. JIT 컴파일은 자바어플리케이션의 실행중 이루어지는 것이므로 사용 리소스나 시간에 제약을 받아서 컴파일시 충분한 최적화를 수행할 수 없다. 그러한 이유로 JIT 컴파일러를 통해 생성한 코드는 일반적으로 높은 품질을 지니지 못한다. 그러나 본 발명에 따른 AOT 컴파일은 라리브러리 또는 어플리케이션 클래스 파일의 배포 이전에 행해지므로 충분한 최적화가 가능하고 결과적으로 JIT 컴파일과는 비교할 수 없는 높은 품질의 머신 코드를 생성할 수 있다. AOT 컴파일을 통해 생성된 코드는 어플리케이션 실행시 JVM에 의해서 후처리 작업을 거쳐야 사용할 수 있는 온전한 코드가 되지만, 이러한 후처리 작업은 JIT 컴파일과 비교하면 아주 간단한 작업이므로 실행중 오버헤드가 적다. 그러므로 본 발명이 적용된 자바 플랫폼이 타겟 기기에서 자바 어플리케이션을 실행할 때는 빠른 속도를 낼 수 있게 된다.

<105> 또한 JIT 컴파일에 비해 램 메모리 요구 사항이 훨씬 낮다. 이는 램 메모리 제약으로 JIT 컴파일을 적용할 수 없는 임베디드 기기 등에도 본 발명이 적용될 수 있음을 뜻한다. 메모리 등의 리소스 제약이 심한 임베디드 기기는 자바 플랫폼을 탑재하더라도 JIT 컴파일 기법을 수용할 수 없어서 인터프리팅 방식을 사용하는데, 인터프리팅 방식은 JIT 컴파일 방식에 비해 어플리케이션 실행속도가 일반저급로 수배 이상 떨어지는 것으로 알려져 있다. 본 발명을 적용한 자바 플랫폼은 JIT 컴파일 등을 사용하는 경우에 비해서도 어느 정도의 성능 향상을 가져올 수 있는데 리소스 제약으로 JIT 컴파일을 사용하지 못하던 임베디드 기기에 적용한다면 상당한 성능 향상을 가져올 수 있다.

【특허청구범위】

【청구항 1】

자바 실행 장치에 있어서,

표준 클래스 라이브러리에 포함된 클래스 파일을 사전 컴파일하여 얻어진 머신 인스트럭션 클래스 파일을 포함하는 확장된 클래스 라이브러리와,

상기 확장된 클래스 라이브러리에 포함된 상기 머신 인스트럭션 클래스 파일 또는 어플리케이션 파일을 실행하는 자바가상머신을 포함하는 것을 특징으로 하는 자바 실행 장치.

【청구항 2】

제1항에 있어서,

상기 머신 인스트럭션은 심볼 참조 정보가 삽입된 오퍼랜드를 포함하는 것을 특징으로 하는 자바 실행 장치.

【청구항 3】

제2항에 있어서,

상기 자바가상머신은 상기 머신 인스트럭션의 오퍼랜드에 삽입된 심볼 참조 정보를 직접 주소로 변환하는 클래스 링커를 포함하는 것을 특징으로 하는 자바 실행 장치.

【청구항 4】

자바 클래스 파일 구조에 있어서,

콘스탄트와 필드와 메소드를 포함하며,

상기 메소드의 메소드 정보는 심볼 참조 정보가 삽입된 오퍼랜드를 포함하는 머신 인스트럭션으로 구성된 코드 속성을 포함하는 것을 특징으로 하는 자바 클래스 파일 구조.

**【청구항 5】**

제4항에 있어서,

상기 메소드 정보는 예외 처리 또는 가비지 콜렉션을 위한 정보를 더 포함하는 것을 특징으로 하는 자바 클래스 파일 구조.

**【청구항 6】**

제4항에 있어서,

상기 심볼 참조 정보는, констан트 풀 심볼 정보, 자바가상머신 내부 심볼 정보, 데이터 블록의 위치 정보 중 어느 하나를 포함하는 것을 특징으로 하는 자바 클래스 파일 구조.

**【청구항 7】**

자바 실행 방법에 있어서,

- a) 표준 클래스 라이브러리에 포함된 클래스 파일을 머신 인스트럭션을 포함하는 확장된 클래스 파일로 미리 사전컴파일하는 단계와,
- b) 상기 확장된 클래스 파일은 머신 인스트럭션을 실행하는 단계와,
- c) 자바 어플리케이션 파일은 JIT 컴파일 또는 인터프리트에 의해 실행하는 단계를 포함하는 것을 특징으로 하는 자바 실행 방법.

**【청구항 8】**

제7항에 있어서,

상기 a) 단계는,

상기 머신 인스트럭션의 오퍼랜드에 심볼 참조 정보를 삽입하는 단계를 포함하는 것을 특징으로 하는 자바 실행 방법.

**【청구항 9】**

제8항에 있어서,

상기 b) 단계는,

상기 머신 인스트럭션의 오퍼랜드에 삽입된 심볼 참조 정보를 직접 주소로 변환하는 단계를 포함하는 것을 특징으로 하는 자바 실행 방법.

**【청구항 10】**

자바 파일을 사전 컴파일하는 방법에 있어서,

자바 클래스 파일 또는 자바 소스 파일을 심볼 참조 정보가 삽입된 오퍼랜드를 포함하는 머신 인스트럭션으로 변환하는 단계를 포함하는 것을 특징으로 하는 자바 파일 컴파일 방법.

**【청구항 11】**

제10항에 있어서,

상기 자바 클래스 파일은 표준 자바 클래스 라이브러리에 포함된 표준 클래스 파일을 포함하는 것을 특징으로 하는 자바 파일 컴파일 방법.

**【청구항 12】**

자바 가상 머신에서 메소드를 실행하는 방법에 있어서,

실행할 메소드의 메소드 정보는 심볼 참조 정보가 삽입된 오퍼랜드를 포함하는 머신 인스트럭션으로 구성된 코드 속성을 포함하는지를 판단하는 단계와,

상기 머신 인스트럭션으로 구성된 코드 속성을 포함하는 경우에, 상기 심볼 참조 정보를 직접 주소로 링크하여 상기 머신 인스트럭션을 실행하는 단계를 포함하는 것을 특징으로 하는 자바 가상 머신에서 메소드 실행 방법.

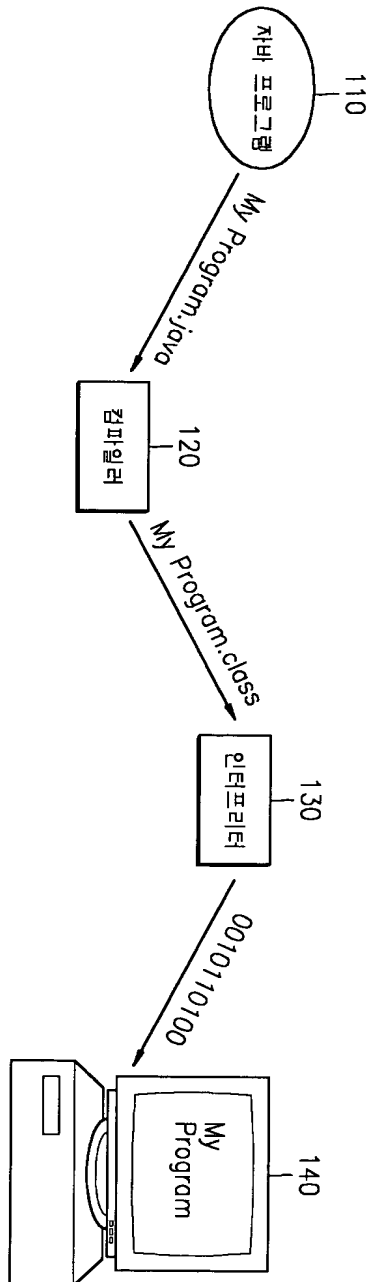
**【청구항 13】**

제12항에 있어서,

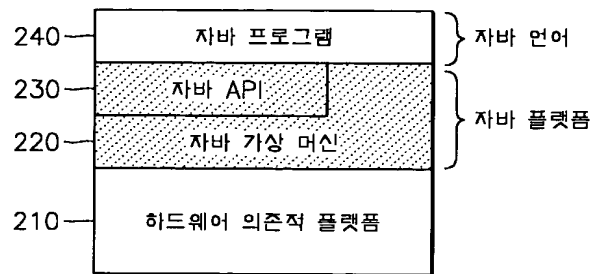
상기 머신 인스트럭션으로 구성된 코드 속성을 포함하지 않는 경우에, 상기 메소드를 JIT 컴파일 또는 인터프리트하는 단계를 더 포함하는 것을 특징으로 하는 자바 가상 머신에서 메소드 실행 방법.

## 【도면】

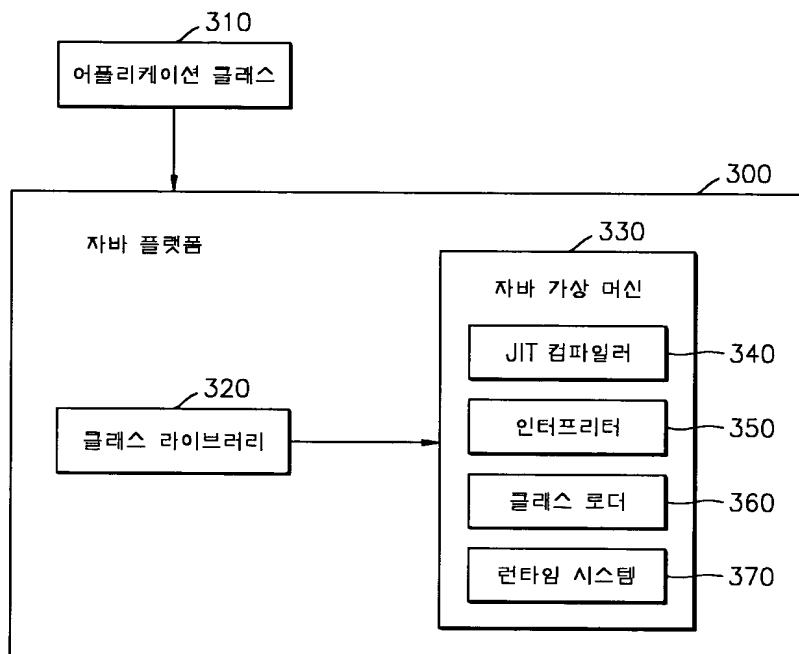
【도 1】



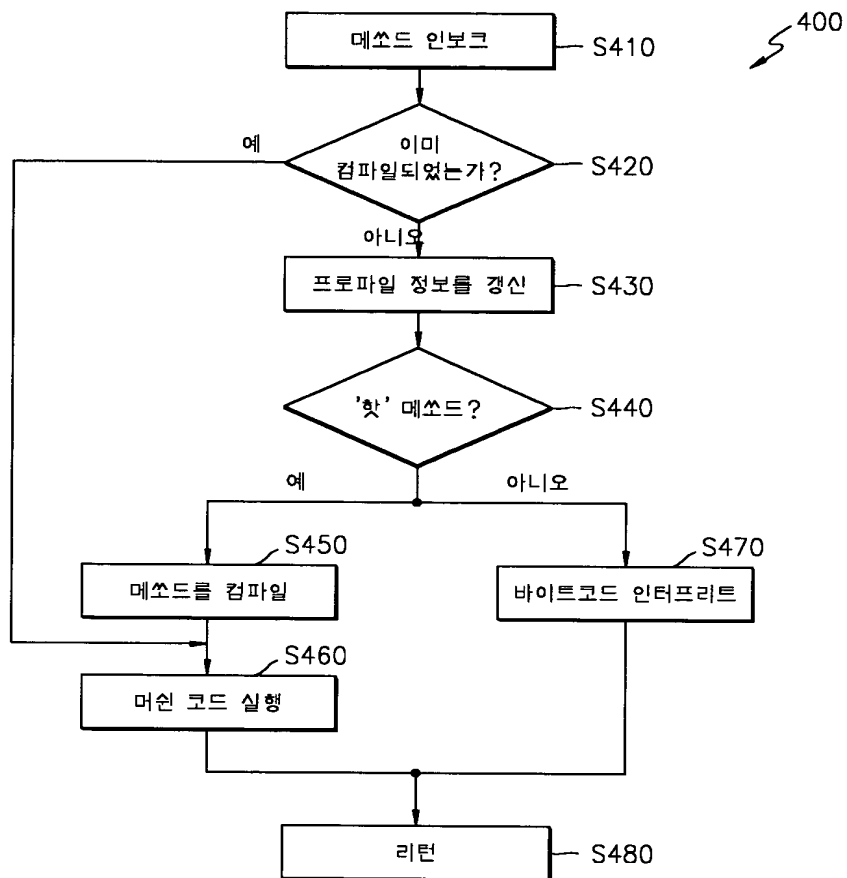
【도 2】



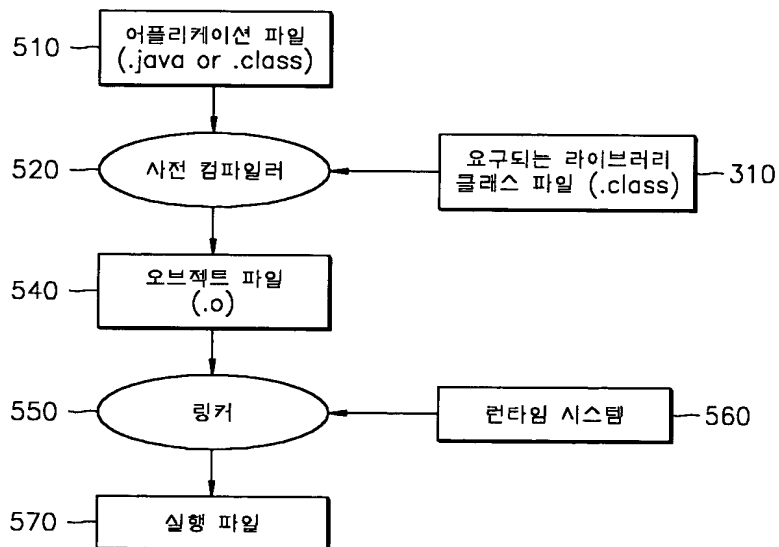
【도 3】



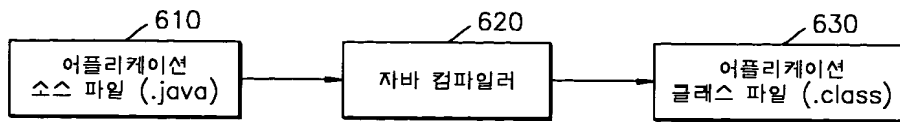
【도 4】



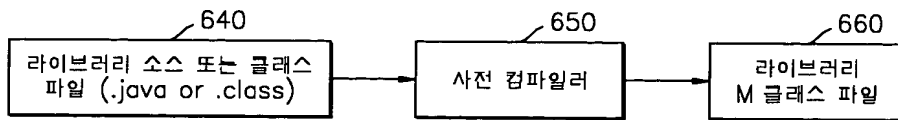
【도 5】



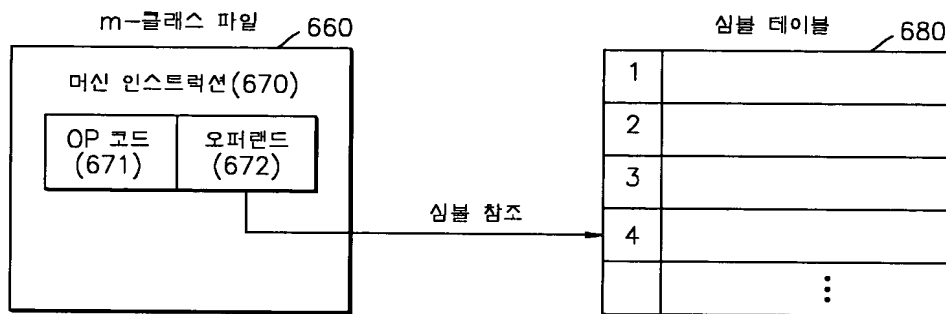
【도 6a】



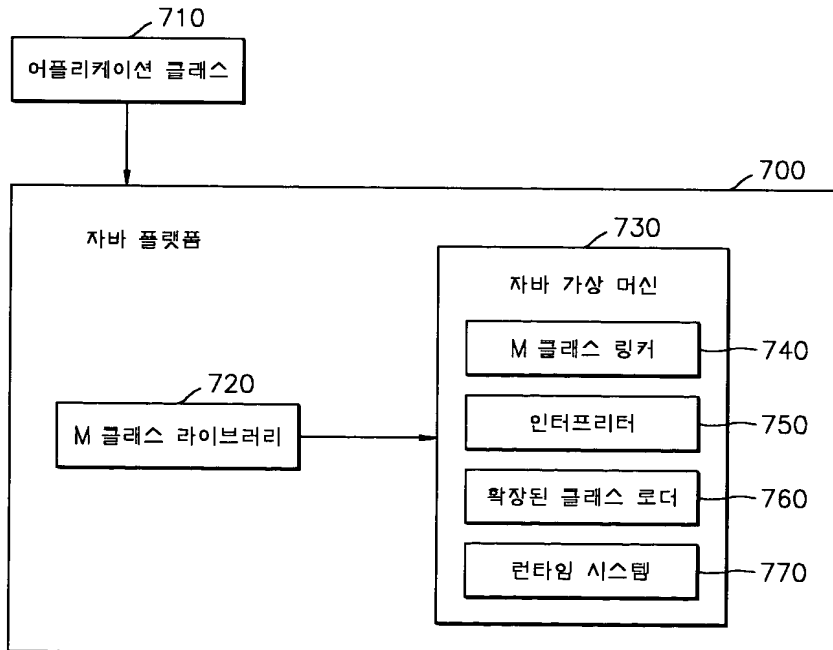
【도 6b】



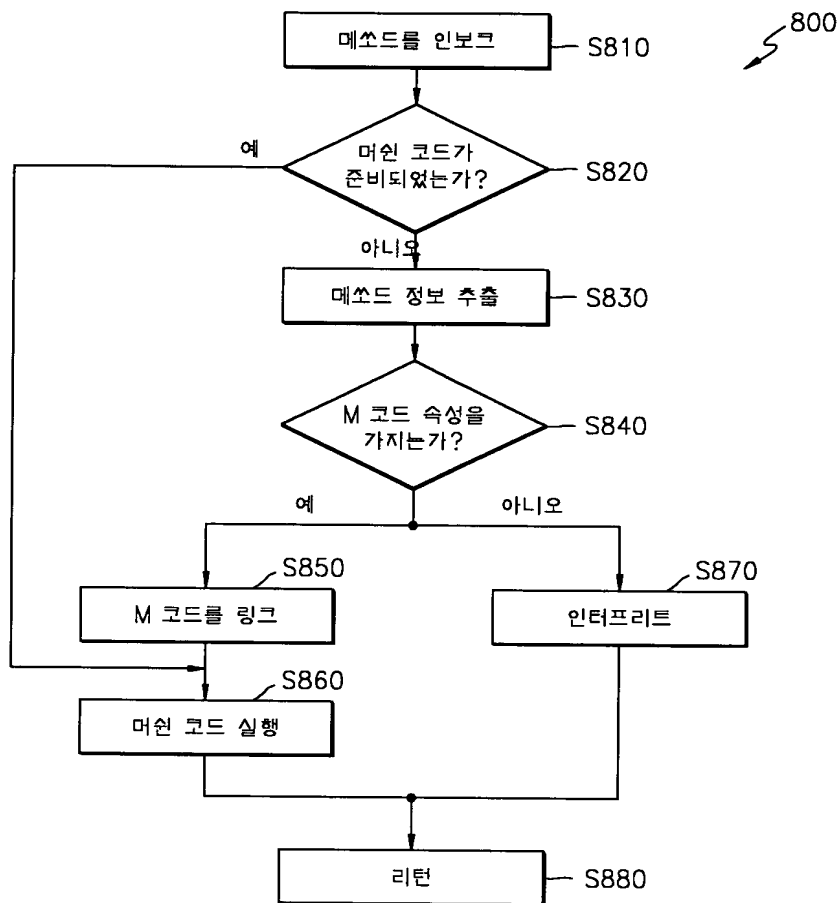
【도 6c】



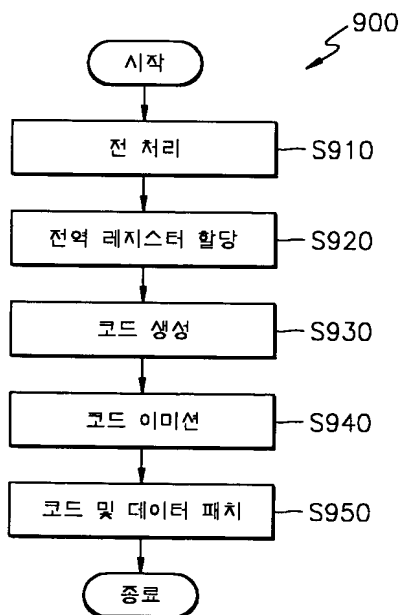
【도 7】



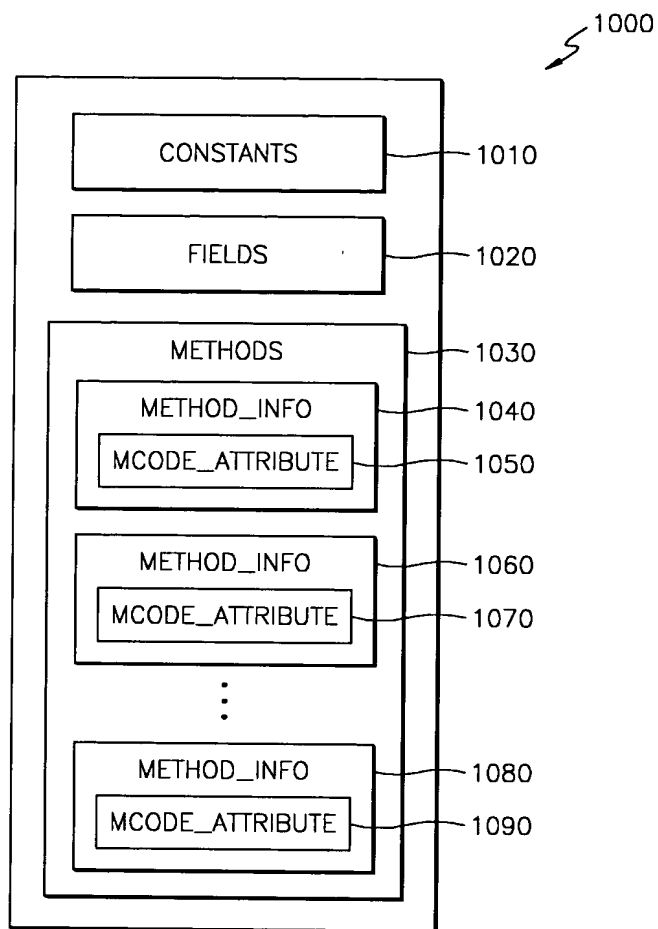
【도 8】



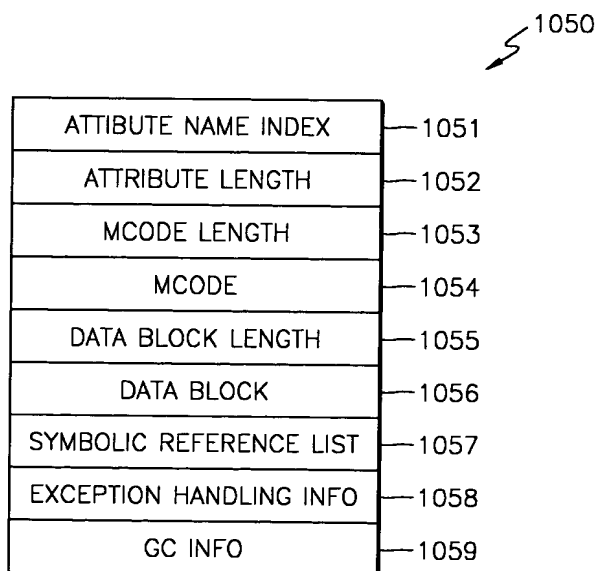
【도 9】



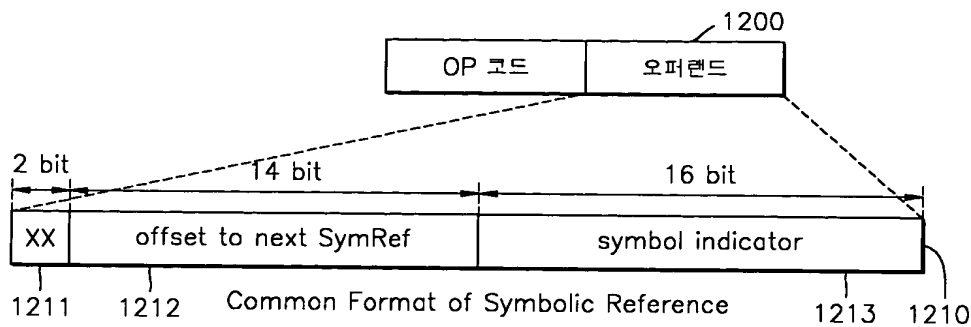
【도 10】



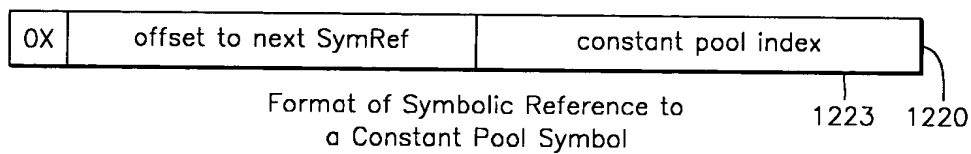
【도 11】



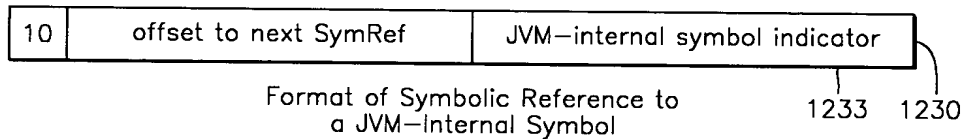
【도 12a】



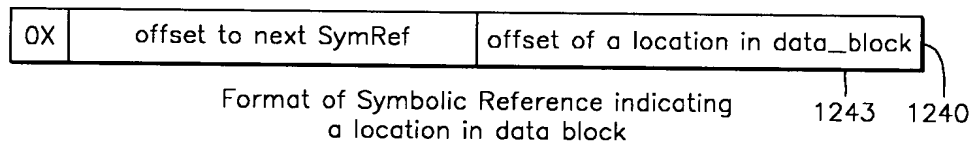
【도 12b】



【도 12c】



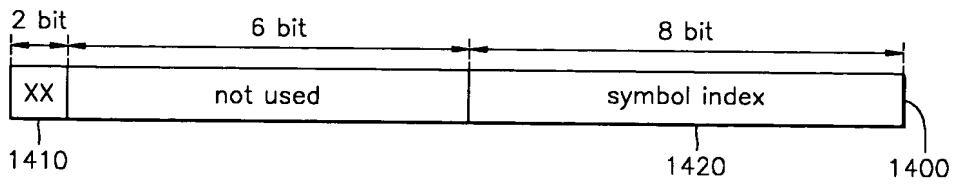
【도 12d】



【도 13】

type	flag (0X)	Meaning
class	00	address of this class
	01	address of the array class having elements of this class type
field	00	address of the location storing the value of this field
	01	(address of the location storing the value of this field) + 4
method	00	address of this method (treat this method as a static method)
	01	address of this method (treat this method as a special method)

【도 14】



【표 15】

Java Platform	Total Time	FLT	FLT/TT (%)	JIT Time	m-Linking Time	JT/TT (%)	MLT/TT (%)	JIT Count	m-Linking Count	JIT Time per method	m-Linking Time per method
orp	4,515,625	3,000,000	66%	703,125	0	16%	0.5%	144	0	4,883	0
m-orp	3,500,000	2,687,500	76%	2,512	187,500	0.7%	5%	1	143	2,512	1,311